Solving Analogies on Words based on Minimal Complexity Transformations

Pierre-Alexandre Murena[1,2*], **Marie Al-Ghossein**[2], **Jean-Louis Dessalles**[2] and **Antoine Cornuéjols**[3]

[1]Helsinki Institute for Information Technology HIIT, Department of Computer Science, Aalto University, Espoo, Finland

[2]LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

[3]UMR MIA-518, AgroParisTech - INRA, Paris, France

pierre-alexandre.murena@aalto.fi, marie.alghossein@telecom-paris.fr, dessalles@telecom-paris.fr, antoine.cornuejols@agroparistech.fr

Abstract

Analogies are 4-ary relations of the form "A is to B as C is to D". When A, B and C are fixed, we call analogical equation the problem of finding the correct D. A direct applicative domain is Natural Language Processing, in which it has been shown successful on word inflections, such as conjugation or declension. If most approaches rely on the axioms of proportional analogy to solve these equations, these axioms are known to have limitations, in particular in the nature of the considered flections. In this paper, we propose an alternative approach, based on the assumption that optimal word inflections are transformations of minimal complexity. We propose a rough estimation of complexity for word analogies and an algorithm to find the optimal transformations. We illustrate our method on a large-scale benchmark dataset and compare with state-of-the-art approaches to demonstrate the interest of using complexity to solve analogies on words.

1 Introduction

An analogy is a Boolean relation of arity 4 that usually reads "A is to B as C is to D" and will be written $A : B :: C : D$ in this paper. It draws a parallel between two *domains*: the *source* domain, consisting of the first two terms A and B, and the target domain, consisting of the last two terms C and D.

Among other applications, analogies have been extensively studied for their role in Natural Language Processing. The fundamental role of analogy in linguistics has been known since [de Saussure *et al.*, 1916]. In his course, the notion of analogy is introduced in the form of a problem: given two forms of a same word (the source) and one form of a second word (the target), the purpose is to induce the second form of the target word. For instance, in German, an analogy could be "setzen:setzte::lachen:x" where x is unknown. This analogy corresponds to the flexion from the infinitive form to the preterit form of the two verbs. The idea of the *saussurian analogy* is then developed in [Lepage and Shin-Ichi, 1996]. Its main principle is applied in several contexts such as machine translation [Lepage and Denoual, 2005], [Lepage and Lieber, 2018], automatic pronunciation [Dedina and Nusbaum, 1991], [Marchand and Damper, 2000] and transliteration [Langlais, 2013].

In this paper, we focus on the question of word inflection (ie. modification of words) and propose a novel method to solve analogical equations on words. A typical application of our method consists of finding the value of x in the equation "solve:solves::get:x". This method is based on the idea that there exists a single transformation rule that applies for both source and target domains and the purpose of which is to describe the transformation from the main form to its inflection. For instance, such a rule to describe English plural would state that letter 's' has to be added at the end of the singular form. Obviously, infinitely many such rules exist. Identifying the relevant rule is often difficult and is a challenge to the existing methods. Our point is to show that, in general, the chosen rule corresponds to the least complex one. In order to quantify the relevant notion of complexity, we present a simple language inspired by the language developed in [Murena *et al.*, 2017] and use it as a proxy in the approximation of Kolmogorov complexity [Li and Vitányi, 2013].

The remainder of this paper is organized as follows. In a first section, we briefly review existing methods to solve analogies, and in particular analogies on words. Then, we introduce our framework of transformation-based analogies and show potential applications of Kolmogorov complexity for the task of solving analogies in this framework. We present next a restricted version of complexity, which is necessary due to the incomputability of complexity. An algorithm is then proposed, followed by some experimental results on benchmark problems. We conclude the paper by a discussion on the performance of our method and on its current limits.

2 Related Works

As presented in the introduction, analogy is a broad research domain and we will present only papers related to solving morphological analogies on words. We exclude semantic analogies on words (king:queen::man:woman). For other aspects of analogy (for instance analogies in knowledge graphs), we refer the reader to reference surveys such

*Contact Author

as [Gentner and Forbus, 2011] or [Prade and Richard, 2014].

A seminal work in the domain of computational analogy is proposed in [Lepage and Shin-Ichi, 1996]. Their model is able to perform operations such as prefixation, suffixation and infixation. The methodology put forward is based on the idea that, given an edit distance $d(.,.)$ between words, an analogy $u : v :: w : x$ is true if and only if $d(u, v) = d(w, x)$, $d(u, w) = d(v, x)$ and $d(v, w) = d(u, x)$. This first computational model is equivalent to the parallelogram rule which is involved in the task of solving of analogies in vector spaces [Rumelhart and Abrahamson, 1973]. A more general formalization of this principle is given with *proportional analogy* [Miclet *et al.*, 2008]. A proportional analogy is a 4-ary relation R on a set X for which the following results hold true for all $u, v, w, x \in X^4$:

- $u : v :: w : x \Leftrightarrow w : x :: u : v$
- $u : v :: w : x \Leftrightarrow u : w :: v : x$
- $u : u :: v : x \Rightarrow x = v$ or $u : v :: u : x \Rightarrow x = v$

The formal link between the solution based on the edit distance and the axioms of proportional analogy is discussed in [Lepage, 2004].

A recent example of a method directly based on the axioms of proportional analogy can be found in [Lepage, 2017]. The idea of the method is to apply these axioms on the number of characters in the four words, on the number of occurrences of each letter, but also on terms of arithmetic relations describing the repetition of common patterns.

An alternative definition of proportional analogy on words is given by [Yvon, 2003]. Based on the definition of a *shuffle* operator $\circ$ (which performs an ordered merging of consecutive subsequences of two words) and of a *complementary set* operator $\setminus$ (which removes the consecutive letters of a first word from the second word), it is shown that x is a solution of $u : v :: w : x$ if and only if $x \in \{v \circ w\} \setminus u$. A consequence of this new characterization is that the set of all possible solutions for an analogy can be obtained based on a finite-state automaton. A development of this original idea was presented with the *alea* algorithm [Langlais *et al.*, 2009] which is based on a random shuffling of the inputs.

Based on the parallelogram rule mentioned above, word embedding techniques have recently emerged. It has been noticed that the word2vec algorithm for word embedding preserves the analogical relations [Mikolov *et al.*, 2013]. The model has been criticized for various limitations (see for instance [Drozd *et al.*, 2016] or [Rogers *et al.*, 2017]). Also, its general methodology is very different from what we propose, since word embedding requires a training step on a large-scale dataset, while we focus on solving analogies based on one observation only. Additionally, analogies can be solved only with words within the training dataset and word2vec cannot handle new words.

We will conclude this short review by a different but strongly related domain based on Hofstadter's microworld [Hofstadter and Mitchell, 1995]. This microworld has been developed as a toy domain covering an exhaustive number of general problems encountered with analogies. Even if the domain consists of letter strings, it differs from the classical framework of morphological analogies by the addition of some structural operations, such as copy or succession operators. The *Copycat* algorithm, focusing on this microworld, is able to solve analogical equations like $abc : abd :: ijk : x$ [Hofstadter, 1984]. It is noticeable that, in most cases, *Copycat* fails at solving morphological analogies.

In an attempt to solve analogies in Hofstadter's microworld, [Cornuéjols and Ales-Bianchetti, 1998] suggested the use of the Minimum Description Length principle, an inductive principle which minimizes the Kolmogorov complexity of the observed analogy. Recently, [Murena *et al.*, 2017] introduced a generative language to evaluate the solutions of analogical equations on Hofstadter's microworlds. The idea of using Kolmogorov complexity to solve analogies is also present in [Bayoudh *et al.*, 2010], which employs it as a measure of distance between concepts for proportional analogies.

The present paper proposes to adapt the ideas of [Murena *et al.*, 2017] to the special case of morphological analogies. However, in contrast to their work, the language we consider is completely agnostic to the alphabetical and morphological structures. Besides, we also provide an algorithm for the automatic search of the solutions of analogical equations.

3 Our Proposition: Minimum Complexity Analogies

3.1 General Notations

We consider a finite alphabet $\mathcal{A}$. We will see later that this alphabet does not need to be known in advance by our method and can be discovered on the fly. Elements of an alphabet are called letters. A word on $\mathcal{A}$ is defined as a finite sequence of letters. The set of words is written $\mathcal{A}^*$. We consider an extension of the alphabet $\bar{\mathcal{A}} = \mathcal{A} \cup \{:, ::\}$ where : and :: are two additional symbols. We define the set of analogies on $\mathcal{A}$ as $An(\mathcal{A}) = \{a : b :: c : d | (a, b, c, d) \in (\mathcal{A}^*)^4\} \subset \bar{\mathcal{A}}$ where the notation $a_1 a_2$ designates the concatenation of a_1 and $a_2 \in \bar{\mathcal{A}}^*$.

Let $Val(\mathcal{A}) \subset An(\mathcal{A})$ be a subset of analogies called *valid analogies*. Given fixed words $(a, b, c) \in (\mathcal{A}^*)^3$, solving the analogy equation $a : b :: c : x$ consists in finding the values of $x \in \mathcal{A}^*$ such that $a : b :: c : x \in Val(\mathcal{A})$.

3.2 Our Framework: Transformation-based Analogies

In the context of grammatical word-transformations, analogies can be restricted to an even more specific expression, following the idea that a language is the result of a cultural evolution which tends to regularize forms (except for the most frequent ones) [Kirby and Hurford, 2002]. We suppose the existence of some grammatical rules which are supposed to apply to the main word. A rule is defined as a function r that can map a word u into a word v. For instance, a rule corresponding to the English plural would consist in adding an 's' at the end of the word. However, a rule does not apply on any word (for instance, the previous rule for the plural would not work for the word 'mouse'). We call the support of the rule the set of words on which the rule applies. The support of rule r is denoted by $Supp(r)$. The valid analogies relative to rule r can be then defined as: $Val_r(\mathcal{A}) = \{a : r(a) :: c : r(c) | a, c \in Supp(r)\}$.

In this paper, we consider a related but slightly different formulation. We aim to describe the base word and its inflection inside each domain *at the same time*. Consequently, we associate each rule r to a function T_r, called *transformation*, which takes some parameters as input and outputs the terms $a : r(a)$ for any possible $a \in Supp(r)$. Due to the nature of the problem, the function T_r needs to be computable and the purpose of solving an analogy is to infer the right transformation. The problem is that infinitely many candidate transformations apply to an analogy. In our paper, we assume that the solution to an analogy is given by the transformation with the shortest coding. A formalization of this idea is given by Algorithmic Information Theory, and in particular by the notion of Kolmogorov complexity.

This assumption is motivated by several results. It has been exposed by [Chater, 1999] that the search for simplicity is a fundamental cognitive principle implied in pattern detection, in memory but also in reasoning. This idea is supported by the experiments proposed in [Murena *et al.*, 2017]: In the context of analogies in Hofstadter's microworld, a majority of participants proposed the solution of lowest complexity. These observations tend to indicate that complexity minimization might be a valid choice to solving analogical equations.

3.3 Kolmogorov Complexity

The notions presented here are classical notions of complexity theory and can be found in [Li and Vitányi, 2013].

In this section, we will denote by $\mathbb{B}$ the binary set $\mathbb{B} = \{0, 1\}$ and $\mathbb{B}^*$ the set of binary sequences.

A function $\phi : \mathbb{B}^* \to \mathbb{B}^*$ is called partial recursive if its output $\phi(p)$ corresponds to the output of a given Turing machine after its execution with input p. When the corresponding machine does not halt for input p, we consider that $\phi(p) = \infty$. With this notation, the function ϕ can be improperly likened to a Turing machine. In this case, the input p is called a *program*.

Complexity $K_\phi(x)$ of a string $x \in \mathbb{B}^*$, relative to a partial function ϕ, is defined as the length of the shortest program p such that $\phi(p) = x$: $K_\phi(x) = \min_{p\in\mathbb{B}^*}\{l(p) : \phi(p) = x\}$, where $l(p)$ represents the length of the string p.

It can be shown that there exists an additively optimal partial recursive (p.r.) function ϕ_0, which means that for any p.r. function ϕ, there exists a constant c_ϕ such that for all $x \in \mathbb{B}^*$, $K_{\phi_0}(x) \leq K_\phi(x) + c_\phi$. Such functions are used to define Kolmogorov complexity. They present in particular invariance properties, which means that the difference between the complexities defined by two distinct universal p.r. functions is bounded. However, it can be shown that Kolmogorov complexity is not computable, and thus cannot be used in practice.

To overcome this, a solution is to consider an upper-bound of Kolmogorov complexity obtained with a non-optimal p.r. function ϕ. This choice is a strong constraint but corresponds to the choice of a bias inherent to any inductive problem.

4 Restricting the Descriptions

In this section, we present our definition of the p.r. function ϕ that we use to evaluate complexity. In order to define this function, we propose a simple description language for character strings and a binary coding of this language. We would like to insist on the fact that this description language is adapted for the description of any character string. The peculiarities of analogies will be assessed in the next section.

4.1 A Simple Language for Texts

We developed a simple description language for character strings in $\bar{\mathcal{A}}$. This language is inspired by the work of [Murena *et al.*, 2017] but differs in several ways. First, our variant does not include any operator. This difference is motivated by the fact that the system should not have access to any prior information, except for copy and paste operations (which is similar to the idea of [Yvon, 2003]). A second difference is an extended use of memory, which allows one to store in memory instructions based on multiple parameters. As we will present later, the number of parameters is automatically detected when invoking an element from memory.

We present here this language in an informal way, that is sufficient to reproduce it. The source code for a Python interpreter is available on the authors' webpage. In order to make explanations clear, we illustrate each concept with a short example. For simplicity purposes, we consider examples where $\mathcal{A}$ is the Latin alphabet. In the instructions, we use the notation `'a'` to designate the letter $a \in \mathcal{A}$.

(1) The language is defined on the alphabet $\Sigma = \bar{\mathcal{A}} \cup \{\texttt{gr}, \texttt{let}, \texttt{mem}, \texttt{?}\} \cup \mathbb{N}$. The Σ-letter `gr` delimits a group of $\mathcal{A}$-letters, the Σ-letter `let` delimits a memory storage and the Σ-letter `mem` corresponds to a memory call.

(2) An instruction is given as a list of comma-separated words. This instruction is executed sequentially. Each character of $\bar{\mathcal{A}}$ is added to the output. For instance, the sequence `'a','b'` will output *ab*.

(3) Some instructions can be stored in memory. A memory storage is delimited by `let`. The content of a memory storage can include parameters, noted `?i` where i is the index of the parameter. For instance, the instruction `let,?0,?0,let` stores a doubling operation.

(4) Memory is structured as a heap. Its content can be accessed using `mem,i` where i corresponds to the depth of the element in the memory. A `mem,i` instruction is followed by a list of arguments. The number of arguments is directly inferred from the number of arguments in the corresponding `let` instruction. For instance, the instruction

```
let, ?0, :, ?1, let, mem, 0, 'a', 'b'
```

outputs $a : b$. It consists of storing a function $(x, y) \mapsto x : y$, calling the last memory storage and applying it on the two characters a and b. However, if the `mem,0` arguments were changed to `mem,0,'a'` only, the syntax would be incorrect, since the corresponding instruction expects two arguments.

(5) The `gr` is used to group some letters together, as a single instance. This is used to define long parameters of a memory call. For instance, after declaring `let,?0,?0,let` calling `mem,0,'a','b','c'` will output *aabc* while calling `mem,0,gr,'a','b','c',gr` will output *abcabc*. The `gr` instruction corresponds to parenthesis in most usual programming languages. In the following examples, we will write the groups as a concatenation for readability purposes: for instance, `gr,'a','b',gr` will be written `'ab'`.

The most important property of our language is that it can generate any possible string on the alphabet $\mathcal{A}$, and consequently any possible analogy, be it valid or invalid. However, despite this complete description ability, the language is trivially not Turing-complete: This can be verified by considering that the halting problem can be solved for it.

4.2 Some Examples

As an illustration, we now propose some examples of instructions that generate language analogies.

apte : inapte :: élu : x	x = inélu	*(Prefixation)*
let,?0,:,'i','n',?0,let,mem,0,'apte',::,mem,0,'élu'		
átír : átírunk :: kitart : x	x = kitartunk	*(Suffixation)*
let,?0,:,?0,'u','n','k',let,mem,0,'átír',::,mem,0,'kitart'		
pati : patti :: olo : x	x = olto	*(Insertion)*
let,?0,?1,:,?0,'t',?1,let,mem,0,'pa','ti',::,mem,0,'ol','o'		
pria : pria-pria :: keju : x	x = keju-keju	*(Repetition)*
let,?0,:,?0,'-',?0,let,mem,0,'pria',::,mem,0,'keju'		
vantut : vanttu :: autopilotit : x	x = autopilotti	*(Reduplication)*
let,?0,?1,'t',:,?0,'t',?1,let,mem,0,'van','tu',::,mem,0,'autopilot','i'		

Figure 1: Results found by our approach for different types of transformations. We also provide the instruction chosen by our program.

In Figure 1, we present some analogical equations of various natures which are successfully solved by our algorithm, and give for each of them the corresponding instruction in the proposed language. For instance, the code of the first example ("apte : inapte :: élu : inélu") reads as follows: The content of the `let` environment describes the domain by coding for the two terms, the base word (the variable `?0`) and its prefixed version (prefix "in" followed by the variable). The remainder of the instruction aims to apply the rule to two possible words: "apte" and "élu".

4.3 Binary Coding of Instructions

In order to define the p.r. function ϕ, we must define a mapping $\mathbb{B}^* \to \mathbb{B}^*$. In our context, two codes (hence two mappings from a given alphabet to $\mathbb{B}^*$) have to be made explicit: the code of the alphabet $\bar{\mathcal{A}}$ and the code of the programs p (ie. the arguments of function ϕ).

In the context of this work, the code $\mathcal{C}_{\bar{\mathcal{A}}} : \bar{\mathcal{A}} \to \mathbb{B}^*$ of the alphabet $\bar{\mathcal{A}}$ can be arbitrarily chosen with the constraint that it is uniquely decodable (which means that for each concatenation $x_1 \dots x_n$ and $y_1 \dots y_n$, the produced binary strings are distinct: $\mathcal{C}_{\bar{\mathcal{A}}}(x_1) \dots \mathcal{C}_{\bar{\mathcal{A}}}(x_n) \neq \mathcal{C}_{\bar{\mathcal{A}}}(y_1) \dots \mathcal{C}_{\bar{\mathcal{A}}}(y_n)$). This choice has no impact on our method since complexity measures the programs' length and not the outputs' length.

In order to propose a code for the instructions, we use the language described above. We propose a code for the corresponding alphabet, as described in Table 1. By construction, the proposed code is prefix (which means that no codeword is the prefix of another codeword), and consequently it is uniquely decodable.

The choice of this code is an ad-hoc choice that can be considered as the parameter of our method. In particular, it is clear that the result of the analogy equations will depend on the choice of the code. The solution proposed in Table 1 is motivated by several ideas.

gr	00	$\mathcal{A}$	111
mem, n	$01^{n+2}0$	**let**	010
:	100	**::**	101
?n	1101^n0		

Table 1: Binary code chosen for the proposed language. Bold values correspond to codewords in the chosen code. The notation 1^n stands for a 1 repeated n times.

A fundamental remark is that instruction words with small description length will tend to be chosen more often than words with larger description length. This might affect in particular the use of memory: if the memory instructions are too costly, the optimal instructions will not use memory and will prefer spelling the analogy directly. For this reason, all letters have the same complexity: This choice is crucial since it gives equal weight to any choice of letter and does not bias the choice of an optimal solution toward instructions that would be imbalanced in terms of displayed letters.

Since the number of variables in `let` parts is *a priori* unbounded, we propose to code all of them with a common prefix `110`. For variable `?n`, this prefix is followed by n times `0` and a final `1`. For instance, the instruction word `?2` will be coded by `110110`. This choice has two advantages: First, it makes the code uniquely decodable and guarantees that an arbitrary number of variables can be used. Secondly, it adds a penalty for using too many variables by adding one bit for each new variable added. This prevents from having too complex instructions in a `let`. The same idea is applied to the memory calls `mem,n`, with the prefix `011`.

Since the code is uniquely decodable, each binary sequence $p \in \mathbb{B}^*$ corresponds to at most one instruction in our language. We define function ϕ such that $\phi(p)$ is equal to the output of the corresponding instruction if p can be decoded into a valid instruction, and $\phi(p) = \infty$ otherwise.

Since our language allows one to describe each word $x \in \bar{\mathcal{A}}^*$ letter by letter, the following proposition holds true:

Proposition 1. *For each word $x \in \bar{\mathcal{A}}^*$, there exists $p \in \mathbb{B}^*$ such that $x = \phi(p)$.*

5 Two Algorithmic Approaches

In this section, we present the algorithm we used to solve the analogy equations by minimizing complexity.

5.1 First Approach: Global Minimization

The most generic complexity minimization method in a general context would consist in considering that the chosen solution for the analogy equation $a : b :: c : x$ is:

$$x^* = \arg\min_x K(a : b :: c : x) \tag{1}$$

where $K(.)$ is the approximation of Kolmogorov complexity as presented in the previous section. This minimization problem alone gives poor results though, because it includes no restriction on the targeted solution: For many analogies, the obtained solution would be empty with this minimization

only. As a solution, we impose a minimal length for the solution as a function of a, b and c. In our experiments, we fixed it to $|c| + |b| - |a|$, taking our inspiration from the works on proportional analogy [Lepage, 2017].

An exhaustive exploration of problem (1) is not possible because it involves the exploration of an exponential number of programs. For instance, the minimum complexity instruction to describe the analogy "rosa:rosam::vita:vitam" is of complexity 117 and it would require exploring all shorter programs, ie. more than 3×10^{35} programs! However, one can notice that it is not necessary to go over the whole exploration space since some instructions are trivially not valid and since an invalid instruction can never be the prefix for a valid one. Consequently, we prune the exploration space based on various implications of the grammar.

To do so, we sequentially explore all instructions by increasing length. The idea is to start with the shortest instructions, discard those that are invalid, and avoid exploring all of the instructions of greater length that have this instruction as prefix. Invalid instructions are those that are syntactically incorrect and that cannot be corrected by extension (for instance, an instruction containing a variable `?0` outside a `let` environment). Invalid instructions also include those yielding results that do not match the known part of the analogy considered. The exploration goes on by increasing the length of the instructions until we find an instruction that appropriately describes the analogy at hand with a solution that verifies the minimal length condition, as mentioned before.

Even with these simplification operations done, the exploration space remains large and consequently, the overall computation time is extremely high. For instance, and with a reasonable implementation in C++[1] run on a machine with one processor Intel Core i7 2.9 GHz and 8G of RAM, it requires a couple of seconds to find a solution for the analogy "rosa:rosam::vita:x" and about 20 minutes for the analogy "orang:orang-orang::burung:x". This difference is explained by several factors such as the length of the words forming the analogy and the patterns observed in words, e.g., repetition of letters or of groups of letters. Note that the exploration space can be processed in parallel as the explored branches are independent from each other, but the results we report here are obtained with a single-threaded process to give to the reader a better sense of the performance of the algorithm.

Aside from the computational complexity of this approach, we would like to present some results related to its performance with regards to the task we are trying to solve. To this end, we randomly select a subset of around 23K examples from a benchmark dataset [Lepage, 2017] and containing a large number of analogies in several languages (a detailed description of this dataset is available in the next section). This first approach presented here performs poorly and the proportion of correct answers we obtain is 27.62%, in contrast with a proportion of 83.44% obtained when using the state-of-the-art method [Lepage, 1998] (more details on this later). This poor performance is due to a large number of programs which break the symmetry between source and target by exploiting one side effect of the language.

[1]The source code is available on the authors' webpage.

5.2 Second Approach: Minimal Transformations

A major drawback of this first method is that it does not exploit the specificity of the chosen framework as presented earlier. Taking these specific features into account leads to a more efficient algorithm to solve analogies on words.

Following the idea that valid grammatical analogies are generated by transformation functions, the research space for programs can be pruned, considering only programs that declare a transformation function and use it to generate both the source and the target. This assumption is motivated by the nature of the desired solution, but it is not equivalent to problem (1). In particular, it is possible that the program of smallest length cannot be interpreted as a transformation.

Since we focus on transformations, ie. on computable functions that describe both the base form and the inflected form of the same word, we need to search for programs with a specific form. These programs must implement the transformation first, then instantiate it for the source problem and finally for the target domain. In our language, this corresponds to searching for programs of the form:

```
let, ... , : , ... , let,
 mem, 0, ... , :: , mem, 0, ...
```

The instruction within the `let` environment corresponds to the description of the transformation. It is called twice, once for the source and once for the target. The exploration task now requires exploring all possible transformations and finding the one of minimal complexity.

The algorithm we propose proceeds by exploring all possible contents for the `let` block. Due to the specific form of the targeted program, only the arguments are needed to instantiate the analogy's description. These arguments can be assessed on the fly while exploring the `let` instruction.

In order to solve the analogy equation $A : B :: C : x$, our algorithm first explores all possible joint descriptions for A and C, which corresponds to the term at the left of the character `:` in the `let` instruction. We sequentially increase the instruction, character by character. If the character is a variable, we store in memory its candidate instantiations. Consequently, this first step yields a list of all possible descriptions, associated with the possible variable instantiations for both A and C. A similar exploration can then be done for the B term, using the already existing memory instantiations. A Python implementation is made available on the authors' webpage.

6 Experimental Results

In this section, we present the experiments we carried out to evaluate our approach.

We first adopt a qualitative perspective and show how our approach addresses different types of transformations. In fact, solving analogies on words requires addressing different types of flectional rules such as prefixation (fini:infini), suffixation (rosa:rosam), insertions (koirani:koiralleni), repetition (burung:burung-burung) and reduplication (puhua:puhuu). Figure 1 shows one example for each type of rule and the solutions generated by our approach for each one of them. These results show that our method is able to solve a large variety of problems, including some that are provably unsolvable by proportional analogy (repetition).

Language	#analogies	NLG_COMP	NLG_PROP	NLG_ALEA
Arabic	165,113	87.18%	**93.33%**	81.91%
Finnish	313,011	**93.69%**	92.76%	78.75%
Georgian	3,066,273	**99.35%**	97.54%	88.42%
German	730,427	**98.84%**	96.21%	95.42%
Hungarian	2,912,310	**95.71%**	92.61%	86.02%
Maltese	28,365	**96.38%**	84.72%	91.84%
Navajo	321,473	81.21%	**86.87%**	78.95%
Russian	552,423	96.41%	**97.26%**	95.46%
Spanish	845,996	**96.73%**	96.13%	94.42%
Turkish	245,721	**89.45%**	69.97%	70.06%
Total	9,181,112	**96.41%**	94.34%	87.93%

Table 2: Proportion of correct answers when solving analogies from the dataset SIGMORPHON'16 using our method NLG_COMP and two state-of-the-art methods NLG_PROP [Fam and Lepage, 2018] and NLG_ALEA [Langlais *et al.*, 2009].

We now evaluate our approach on a large dataset of analogies. We base our work on one of the largest datasets available for solving analogies on words that we denote by SIGMORPHON'16 and that is presented in [Lepage, 2017]. This dataset is from the Track 1 Task 1 of SIGMORPHON 2016 Shared Task [2]. Analogies were extracted from the original dataset proposed in the context of the Shared Task. The data includes ten languages (Table 2). As mentioned in [Cotterell *et al.*, 2018], considering these different languages covers a variety of structures such as prefixes and consonant harmony in Navajo, suffixes in Turkish, templatic morphology in Arabic, and vowel harmony in Hungarian and Finnish.

The original dataset [3] contains around 65M analogy questions. It includes four questions for each original analogy, where each of the analogy's terms becomes the answer for each question. We select one analogy for each set of four questions and keep unique analogies by removing duplicate questions. We thus obtain a dataset of around 9M analogies.

To assess the performance of our approach, we compare it to the two state-of-the-art methods used for solving analogies on words and based on proportional analogy: NLG_PROP [Fam and Lepage, 2018] and NLG_ALEA [Langlais *et al.*, 2009]. Results are presented, separately for each language, in Table 2.

The results show that our approach, NLG_COMP, outperforms NLG_PROP and NLG_ALEA on the whole set of analogies considered. Looking at the results obtained per language, NLG_COMP outperforms NLG_PROP for Finnish, Georgian, German, Hungarian, Maltese, Spanish, and Turkish, but performs worse than NLG_PROP for Arabic, Navajo, and Russian. While NLG_COMP outperforms NLG_ALEA on all languages, NLG_PROP performs better than NLG_ALEA except for Maltese and Turkish. We note that the different languages are not equally represented in the original dataset SIGMORPHON'16. It is worth mentioning that applying NLG_ALEA usually generates several candidate solutions, but we consider here the most frequent solution given our setting.

General cases of transformations on which our approach cannot perform well include irregularities (work:worked::go:*went*), transformations of the radical (tori:torilla::katu:*kadulla*) or conditional changes (rules conditional to the radical of the word) such as vowel harmony (hat:hatban::egy::*egyben*). While these cases cannot either be solved by state-of-the-art methods, some other transformations that can be solved by NLG_COMP but not by NLG_PROP and NLG_ALEA are not included in SIGMORPHON'16 (see for example the *repetition* transformation in Figure 1).

We also compared our approach to word embedding techniques. For this purpose, we used pre-trained word embeddings [Grave *et al.*, 2018]. We could observe a very poor performance, with a proportion of correct answers ranging from about 17% (for German) to less than 0.1% (for Maltese). These results were expected: Not only the four words of the analogy have to be present in the corpus on which the embeddings have been trained, but also in a statistically significant proportion so that correct information can be acquired. This is obviously not the case for most forms. These results tend to indicate that using word embeddings is more effective to solve semantic analogies than grammatical analogies.

To conclude, we point out a major advantage of our solution, which is its *interpretability*. The output of the algorithm is not only the inferred fourth term of the analogy, but also the description that justifies this choice. Such a description could be exploited by an agent to explain its results to a user.

7 Conclusion

In this paper, we presented a novel algorithm for solving morphological analogies on words. This approach differs from most state-of-the-art methods in the fact that it does not obey the axioms of analogical proportion. It follows a principle of complexity minimization where complexity has to be understood as the length of the shortest description of the analogy. Although the idea of using complexity as a tool for solving analogical equations has already been mentioned, our paper is the first one to actually present an algorithm able to solve this problem and to validate this assumption on large datasets.

Our algorithm explores a very large space of programs in order to determine the minimal instruction that generates a correct analogy. In order to bypass the difficulty of the task, we focus on instructions of a grammatically plausible form. We could validate our method on the benchmark dataset SIGMORPHON'16 on which we obtain competitive results. In addition, our algorithms yields a description of its answer, which is a major difference with existing methods in terms of interpretability and explainability.

In this paper, we considered that each target is associated with one source, which reduces the problem to finding the fourth term of the analogy. To be more realistic, the source should be found by the agent in a list of already known forms, such as in case-based reasoning. This issue should pave the way to more realistic models for language acquisition.

Acknowledgments

This work was supported by the Academy of Finland Flagship programme: Finnish Center for Artificial Intelligence, FCAI. We would like to thank the reviewers for their helpful comments.

[2] http://ryancotterell.github.io/sigmorphon2016/

[3] http://lepage-lab.ips.waseda.ac.jp/en/projects/kakenhi-15k00317/

References

[Bayoudh *et al.*, 2010] Meriam Bayoudh, Henri Prade, and Gilles Richard. A kolmogorov complexity view of analogy: From logical modeling to experimentations. In *SGAI Conf.* Springer, 2010.

[Chater, 1999] Nick Chater. The search for simplicity: A fundamental cognitive principle? *The Quarterly Journal of Experimental Psychology: Section A*, 52(2), 1999.

[Cornuéjols and Ales-Bianchetti, 1998] Antoine Cornuéjols and Jacques Ales-Bianchetti. Analogy and Induction : which (missing) link? In *Workshop "Advances in Analogy Research : Integration of Theory and Data from Cognitive, Computational and Neural Sciences"*, 1998.

[Cotterell *et al.*, 2018] Ryan Cotterell, Christo Kirov, John Sylak-Glassman, Géraldine Walther, Ekaterina Vylomova, Arya D McCarthy, Katharina Kann, Sebastian Mielke, Garrett Nicolai, Miikka Silfverberg, et al. The conll–sigmorphon 2018 shared task: Universal morphological reinflection. In *CoNLL Shared Task*, 2018.

[de Saussure *et al.*, 1916] F. de Saussure, C. Bally, A. Sechehaye, and A. Riedlinger. *Cours de linguistique générale*. Payot, 1916.

[Dedina and Nusbaum, 1991] Michael J Dedina and Howard C Nusbaum. Pronounce: a program for pronunciation by analogy. *Computer speech & language*, 5(1):55–64, 1991.

[Drozd *et al.*, 2016] Aleksandr Drozd, Anna Gladkova, and Satoshi Matsuoka. Word embeddings, analogies, and machine learning: Beyond king-man+ woman= queen. In *COLING*, 2016.

[Fam and Lepage, 2018] Rashel Fam and Yves Lepage. Tools for the production of analogical grids and a resource of n-gram analogical grids in 11 languages. In *LREC*, 2018.

[Gentner and Forbus, 2011] Dedre Gentner and Kenneth D Forbus. Computational models of analogy. *Wiley interdisciplinary reviews: cognitive science*, 2(3), 2011.

[Grave *et al.*, 2018] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *LREC*, 2018.

[Hofstadter and Mitchell, 1995] Douglas Hofstadter and Melanie Mitchell. Fluid concepts and creative analogies. chapter The Copycat Project: A Model of Mental Fluidity and Analogy-making. 1995.

[Hofstadter, 1984] Douglas Hofstadter. The Copycat Project: An Experiment in Nondeterminism and Creative Analogies. AI Memo 755, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1984.

[Kirby and Hurford, 2002] Simon Kirby and James R Hurford. The emergence of linguistic structure: An overview of the iterated learning model. In *Simulating the evolution of language*. Springer, 2002.

[Langlais *et al.*, 2009] Philippe Langlais, François Yvon, and Pierre Zweigenbaum. Improvements in analogical learning: application to translating multi-terms of the medical domain. In *EACL*, 2009.

[Langlais, 2013] Phillippe Langlais. Mapping source to target strings without alignment by analogical learning: A case study with transliteration. In *ACL*, volume 2, 2013.

[Lepage and Denoual, 2005] Yves Lepage and Etienne Denoual. Purest ever example-based machine translation: Detailed presentation and assessment. *Machine Translation*, 19(3-4), 2005.

[Lepage and Lieber, 2018] Yves Lepage and Jean Lieber. Case-based translation: First steps from a knowledge-light approach based on analogy to a knowledge-intensive one. In *ICCBR*. Springer, 2018.

[Lepage and Shin-Ichi, 1996] Yves Lepage and Ando Shin-Ichi. Saussurian analogy: a theoretical account and its application. In *COLING*, 1996.

[Lepage, 1998] Yves Lepage. Solving analogies on words: an algorithm. In *COLING-ACL*, volume I, 1998.

[Lepage, 2004] Yves Lepage. Analogy and formal languages. *Electronic notes in theoretical computer science*, 53, 2004.

[Lepage, 2017] Yves Lepage. Character-position arithmetic for analogy questions between word forms. In *ICCBR (Workshops)*, 2017.

[Li and Vitányi, 2013] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[Marchand and Damper, 2000] Yannick Marchand and Robert I Damper. A multistrategy approach to improving pronunciation by analogy. *Computational Linguistics*, 26(2), 2000.

[Miclet *et al.*, 2008] Laurent Miclet, Sabri Bayoudh, and Arnaud Delhay. Analogical dissimilarity: definition, algorithms and two experiments in machine learning. *J. Artif. Intell. Res.*, 32, 2008.

[Mikolov *et al.*, 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[Murena *et al.*, 2017] Pierre-Alexandre Murena, Jean-Louis Dessalles, and Antoine Cornuéjols. A complexity based approach for solving hofstadter's analogies. In *ICCBR (Workshops)*, 2017.

[Prade and Richard, 2014] Henri Prade and Gilles Richard. *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548. Springer, 2014.

[Rogers *et al.*, 2017] Anna Rogers, Aleksandr Drozd, and Bofang Li. The (too many) problems of analogical reasoning with word vectors. In **SEM*, 2017.

[Rumelhart and Abrahamson, 1973] David E Rumelhart and Adele A Abrahamson. A model for analogical reasoning. *Cognitive Psychology*, 5(1), 1973.

[Yvon, 2003] François Yvon. Finite-state transducers solving analogies on words. *Rapport GET/ENST&LTCI*, 2003.